Q10： 正则表达式匹配 MD:字符串

Q5： 最长回文子串。 MD:字符串

Q32 . 最长有效括号 一个有效的字符串一定是 从左到右扩展的子串，都是满足 左括号数量 >= 右括号数量的。且该串本身 左括号数量 = 右括号数量

1. 方法1：使用dp[i][j] = k 表示 i~j的子串，k = 左括号-右括号数量。当k = -1时，表示i~j一定非法，且以i开头，jj>j的的其他串也都是非法的。k > 0 则表示后序还有可能是合法的。该方法是最只直观的方法。但是时间复杂度 o(n^2)太慢了。
2. **双指针** 还是左括号数量m >= 右括号数量n的的，进行双向 遍历 出现m = n就保留最大， m < n就 mn置0.从下一个位置从新开始数。**反向再找一遍** 。
3. **动态规划** dp[i] = k : 表示以下标 i 字符结尾的最长有效括号的长度 。
   1. 以'('结尾的最长长度，必定是0.所以我们只对')'结尾的位置dp即可
   2. 根据括号可能的形式：() ; ()() ; ((())) ; ()((()))来看dp的递推关系。
   3. (), 则dp[1] = 2; ()(),则dp[3] = dp[1]+2;
   4. ((())), 则dp[5] = dp[4] + 2 ,因为若i-1的位置不是'('说明以位置i-1结尾是一个有效的括号序列。
   5. ()((())), 则dp[5] = dp[4] + ? 我们要获取上一个有效括号序列才行。

   if (s.charAt(i) == ')') { if (s.charAt(i - 1) == '(') { dp[i] = (i >= 2 ? dp[i - 2] : 0) + 2; } else if (i - dp[i - 1] > 0 && s.charAt(i - dp[i - 1] - 1) == '(') { dp[i] = dp[i - 1] + ((i - dp[i - 1]) >= 2 ? dp[i - dp[i - 1] - 2] : 0) + 2; } maxans = Math.max(maxans, dp[i]); }

Q53 : **最大子序和** : 很心痛的一道题

1. 动态规划： f(i) : 以ai结尾的最大子序和。 f(i) = max(f(i-1)+ai, ai); 然后dp压缩，空间使用o(0)
2. 滑动窗口： 假设当前窗口和为sum, 最终结果为ans, 若当前sum <= 0 则如果对其继续扩张，sum对后边反而是负担，不会有贡献。所以舍弃该窗口，然后sum = ai, ans = max(ans,sum) 只有当sum > 0 时对sum扩张才有意义。
3. 如何舍弃已有窗口: 看贡献值。 贡献值是一个很常用的窗口收缩策略。

Q62 : 不同路径 **杨辉三角**

1. 每个位置的路径总数 = 该位置左边的路径总数 + 该位置上边的路径总数
2. 其实就是爬楼梯题的变行，这个是二维的。

Q64 : 最小路径和

1. 左上角到右下角，只能2方向搜索。 这一点给了我们通过迭代dp的机会。

2. 假设该题是4方向的，起点和终点不是在角落，那么要迭代dp就不行了，因为填表顺序太复杂，各个方向都有依赖。但是可以通过递归记忆化来解决。递归记忆化就是为了解决这种计算顺序无法掌握的情况。

   dp[i][j] = min(dp[i+1][j],dp[i][j+1])+m[i][j];

Q91：解码方法 12335323131332 -> abaasdasd 数字到字符的映射。有多少种解码方法。

1. 其实就是爬楼梯 的改编。

```
编码规定。
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

2. 给出一个数字字符串，问： 有多少种解码的方式。 12 可以理解为 1,2解码为AB 也可以理解为 12解码为L

3. 使用dp[i]:表示以i结尾的当前串，有多少种解码方式。 则dp[i]只和dp[i-1],dp[i-2]有关系。

   1. 若 num[i] = 0, 则num[i-1]只能是1,或者2. 那么dp[i] = dp[i-2]; 因为最后两位一定是10或者20. 是死的。

   2. 若 num[i-1] = 1 或者 (num[i-1] = 2 && 0 < num[i] < 7). 那么dp[i] = dp[i-2] + dp[i-1],即位置i是否要和i-1合并。

   3. 剩下的情况一定不能合并，所以 dp[i] = dp[i-1];

   dp[i-2] : pre dp[i-1] : cur for (int i = 1; i < s.length(); i++) { int tmp = curr; if (cs[i] == '0') if (cs[i - 1] == '1' || cs[i - 1] == '2') curr = pre; else return 0; else if (cs[i - 1] == '1' || (cs[i - 1] == '2' && cs[i] >= '1' && cs[i] <= '6')) curr = curr + pre; else curr = curr; //可以省略 pre = tmp; }

总共6题股票问题

1. money[i]: 第i天结束时，持有钱的数目。 hold[i]: 第i天结束时，持有股票的数目。

2. 卖出：+prices[i] ,买入：-prices[i]

3. 没有冷冻期，都是可以当天无限次买入卖出。

4. 手续费总是在卖出时扣除。

Q121：买卖股票的最佳时机I： **只能买入和卖出一次。**

1. 使用滑动窗口， left < right, 然后left永远指向right之前的最小值。 只有在right遇到更小值时，left移动到right的位置,这样就可以得到数组中，求得 num[right] - num[left] 的最大值。

Q122：买卖股票的最佳时机 II： 必须卖出后，才能重新买入。 **可以多次买卖股票**

股票可以在同一天买入和卖出

1. 贪心：只要价格高于买入价格， 就卖掉。假设三天的价格是 7 8 9 买入7，卖出8，再买入8，卖出9 总差价就是2. 等价于7买入，9卖出。
2. 只要今天比昨天高，就计入利润。

Q123： 买卖股票的最佳时机 III : **最多做两笔交易**

股票可以在同一天买入和卖出：

```
                int buy1 = -prices[0], sell1 = 0;
                //注意buy2的初始化，在第一天，我们就可以进行第二次交易，
                // 假设当前股价为7,第一次买入7，第一次卖出7，第二次买入7，所以收益是
-7.
                int buy2 = -prices[0], sell2 = 0;

                for (int i = 1; i < n; ++i) {
                    // 之前就已经买入第一次，持有股票，当天买入第一次
                    buy1 = Math.max(buy1, -prices[i]);
                    // 之前卖出第一次，一直持有现金。 或者今天卖出，持有股票
                    sell1 = Math.max(sell1, buy1 + prices[i]);
                    buy2 = Math.max(buy2, sell1 - prices[i]);
                    sell2 = Math.max(sell2, buy2 + prices[i]);
                }
                return sell2;
```

1. 在计算sell1的时候，为什么要使用刚刚更新过的buy1呢？？？ 这就是巧妙的地方，我们允许今天买入的股票，在今天卖出。
2. 在计算buy2的时候， 我们又使用刚更新过的sell1, 这就表明，我们今天卖出了， 但是还可以继续买入。sell2也是同理。
3. 那么最后我什么返回sell2呢？ 一定是做了两笔交易，才能取得最大利润吗?
4. 不： 得益于上边的设定， 我们总是能得到当天完成两笔交易时取得的最大利润。 因为我们能同一天，完成一笔不赔也不赚的交易。这就是，不设置冷冻期的好处。

Q188 : 买卖股票的最佳时机 IV： **最多做K笔交易**

1. Q123的增强。 变为buy[i], sell[i],即可

Q714 : 买卖股票的最佳时机含手续费 : 不限制交易次数，无冷冻期，有手续费。

1. 我们设定 **在卖出时，扣除手续费** 这样这道题就和 **Q122** 一样了。

Q309 : 最佳买卖股票时机含冷冻期： 不限制交易次数，但是卖出股票后，无法在第二天买入。

1.这题因该分3个状态： 1. 持有股票 2. 不持有股票，并且处于冷冻期的收益， 3.不持有股票，不在冷冻期的最大收益。

```
        f[i][0] = Math.max(f[i - 1][0], f[i - 1][2] - prices[i]);
        // 要持有股票，只能是从已经持有股票，或者从不在冷冻期转来
        f[i][1] = f[i - 1][0] + prices[i];
        // 如果今天处于冷冻期，说明是昨天卖了股票。
        f[i][2] = Math.max(f[i - 1][1], f[i - 1][2]);
```

Q131 : 分割回文串 : 对串分割，要求所有子串都是回文串，返回所有方案。

1. 首先，我们通过dp获得所有子串是否是一个回文串。dp[i][j] = true 则子串ij为回文串。
2. 通过回溯的 begin写法， 以当前位置为起点，准备分割下一个回文串。终点有很多选择。只要dp[i][j] = true即可。
3. 当cur == len时，说明已经分割完毕。存入list即可。

Q139 : 单词拆分 s = "leetcode", wordDict = ["leet", "code"] wordDict不重复，但是每个单词可以重复使用。问：是否可以分割完毕。

1. 使用动态规划： dp[i]：0到i是否可以分割完毕。
2. 转移方程： if(dp[j]) dp[i] = set.contains(s.substring(j+1, i+1)); 如果dp[j] = true,且j+1到i在集合种，那么可以划分成功。
3. **单词拆分II** : 这个题不是求是否可以划分，而是得出所有的划分结果，所以动态规划就不行了， 要使用回溯法递归了。但是可以参考回文串的划分，先使用动态规划得到可划分区域 dp[i] 在dp[i]可划分的位置进行切割。其余位置直接跳过。

Q152 乘积最大子数组 当数组有0的时候，前缀积是没有意义的。

1. 最容易想到的一个方式是，dp[i][j],表示i~j的乘积， 然后使用max在dp中找最大值。时间o(n^2)
2. 是两个dp， max记录以i结尾的最大子数组。min记录以i结尾的最小子数组。min就可以保留负值了。
3. 最大不一定是正数，最小不一定是负数。

```
                    if(nums[i] == 0){
                        max = min = 0;
                    }else if(nums[i] > 0){
                        int ma = max, mi = min;
                        // 前一个最大的 可能也是 负数
                        max = Math.max(ma*nums[i], nums[i]);
                        min = Math.min(mi*nums[i], nums[i]);
                    }else{
                        int ma = max, mi = min;
                        max = Math.max(mi*nums[i], nums[i]);
```

```
                        min = Math.min(ma*nums[i], nums[i]);
                    }
```

4. 当num > 0 以i结尾的最大子数组，只可能是从以i-1结尾的最大子数组，或者子数组是他本身转化来的。

5. 当num < 0 以i结尾的最大子数组, 只可能是从以i-1结尾的最小子数组，或者子数组是他本身转化来的。最小子数组也同理。

打家劫舍

Q198 : 打家劫舍 相邻两家同时被偷，会报警。钱数：[1,2,3,1] 最多偷 1+3 = 4

1. dp[i]: 表示前i家最多可以偷多少。 则第i家的转移有两种

```
            第i家不偷， dp[i] = dp[i-1]
            第i家偷  , dp[i] = dp[i-2] + nums[i]
            那么两种情况取最大值，即可。
            可以用滚动数组两个变量保存 i-1,i-2即可。
```

2. 模仿股票问题，使用双dp数组， **比较简单的设法。我觉的也会比较通用。**

```
             get[i] : 如果偷第i家，前i家最多偷多少
             no_get[i] : 不偷第i家，前i家最多偷多少。
             那么转移方程是
             get[i] = no_get[i-1]+nums[i] // 昨天只能是不偷转移来的。
             no_get[i] = max(no_get[i-1], get[i-1]) // 昨天可能也没偷，也可能偷
了。
```

Q337 : 打家劫舍 III 二叉树的结构，直接相连的两家被偷，就会报警。

1. 二叉树结构，用数组的话，不好掌握计算顺序。所以使用记忆化递归。 记住：get,no_get两种情况可以偷多少。

2. 然后向上层返回，父节点根据子节点的情况再向上返回。所以使用递归的后序方式。

```
        int[] l = dfs(node.left);
        int[] r = dfs(node.right);
        int selected = node.val + l[1] + r[1];
        int notSelected = Math.max(l[0], l[1]) + Math.max(r[0], r[1]);
        return new int[]{selected, notSelected}

        注意：当前结点不偷，则两个子节点都偷，不一定会取得最大值。
```

背包

Q279 完全平方数 13 = 9 + 4 用最少的完全平方数使得其和为 n。

1. 是一个数论问题，但是数论解法太高深了，看不懂。 所以看作背包问题比较好。即k这个数的平方要不要背的问题。

```
        dp[i] = Math.min(dp[i], dp[i - k*k]+1);
```

Q322 : 零钱兑换 完全背包问题。

```
            for(int i = 1; i <= amount; i++){
                for(int j = 0; j < coins.length && coins[j] <= i; j++){
                    dp[i] = Math.min(dp[i], dp[i-coins[j]] + 1);
                }
            }

            因为可以重复选同一种硬币，说明，每次选硬币时，他可以用每一个硬币。只要该硬币
面值 <= i即可。
```

Q416 : 分割等和子集： 是否可以把数组分为两个子集，使得其和相等。**好题**

1. 0-1背包的 硬币问题，是否存在一种选择，使用这些硬币凑够 sum/2

```
    dp[i][j] = dp[i-1][j] | dp[i-1][j-nums[i]];   //只依赖于上一行的 01背包问
题。
    如果不选第i个数，那么前i个数中拼为j,的状态就和上一行的状态相同。
    选第i个数字，那么 若前 i-1个数字，能拼为j-nums[i]，那么前i个数字，就能拼为
j.

    dp[0] = true;
    for(int i = 1; i <= nums.length; i++){
        for(int j = target; j >= nums[i-1]; j--){
            dp[j] = dp[j] | dp[j-nums[i-1]];
        }
    }
    return  dp[target];

    // i-1: 下标是i-1的物品，是 第i个物品。
    //
```

Q494 目标和 nums: [1, 1, 1, 1, 1], S: 3 给数组中每一个元素 +或者-号，使得整个数组的和 为S。 返回方法数。 1. 初看是回溯法，子集生成，每次都选出的集合为正集合。然后最后算看和是否是 S。 回溯法太慢了。 2. 使用

动态规划。 dp[i][j] = k: 表示前i个数字，赋予符号，最终凑够和为j ， 总共有k种方法。 当dp[i-1][]确定时， 第i个物品有两种方案，给正号，或者给 负号。 dp[i][j] = dp[i-1][j+nums[i]] + dp[i-1][abs(j-nums[i])]
假设 j = 5 nums[i] = 2
找 j-nums[i] = 3 的方案数。 [3],+2 = 5 找 j+nums[i] = 7 的方法数 [7],-2 = 5 假设 j = 5 nums[i] = 6 找 j+nums[i] = 11 的方案数 [11],-6 = 5 找 j-nums[i] = -1 的方案数 [-1],6 = 5 但是问题是 dp数组下标 >=0 , 不可能存在 j-nums[i] < 0 j-nums[i]的方案数， 和 nums[i] - j的方案数 是一样的。 只需要把 nums[i] - j的方案中所有符号逆置 就得到了 j-nums[i] ,所以方案数 是一样多的。

testQ1 凑够目标和y的方案数。输入：y,k, 可使用 1...y, x1+x2+..+xn = y xi != xj && xi%k != 0

1. 背包容量为 y, 所以 硬币最大值就是 y. 再大就没有意义.
2. dp[i][j]: 前 i个数，装到容量为j的背包中的方案数

```
dp[i][j] = dp[i-1][j] + dp[i-1][j-i]
1. 若不使用i这个数，那么就需要通过前i-1个数凑够j，方案数为 dp[i-1][j]
2. 若使用i这个数，那么就需要前i-1个数凑够 j-i，方案数为 dp[i-1][j-1]
3. 对于 i%k==0的i这个值，这一行都是非法的，下边也不能依赖，所以上边二维情况下，不能直接使用dp[i-1].而是通过dp[prerow]记录上一行的位置。
```

3. 初始化：dp[0] = 1; 前0个物品放在容量0的背包，方案数是1. 前0个物品放在容量非0的背包，方案数为0.

```
dp[0] = 1;
for(int i = 1; i <= y; i++){
    // 当i%k==0,说明没有 i这个物品。
    if(i % k == 0) continue;
    for(int j = y; j >= i; j--){
        dp[j] = dp[j] + dp[j-i];
    }
}
return dp[y];
```

01背包模板

```
    String[] line_1 = in.readLine().split(",");
    String[] line_2 = in.readLine().split(",");
    int len = line_1.length;
    int[] w = new int[len+1];
    int[] v = new int[len+1];
    // 物品要从 **第一个下标** 开始存放
    for(int i = 1; i <= len; i++){
        w[i] = Integer.parseInt(line_1[i-1]);
        v[i] = Integer.parseInt(line_1[i-1]);
    }
```

```
    dp定义:  f[i][j] = max( f[i - 1][j], f[i - 1][j - w[i]] + v[i])
             f[i][j]表示前i件物品放入一个容量为j的背包可以获得的最大价值

    for (int i = 1; i <= n; i++)
        for (int j = V; j >= w[i]; j--)
            f[j] = max(f[j], f[j - w[i]] + v[i]);

    i从 0到 n. 共 n+1行。是物品的个数+1
    j 从 0到 target, 共 target+1列

    // 若 w,v从0开始放物品  反正i就和wv有关系。 i转够n圈即可。但这只适用于可以一维度的情况下。即
    // dp不需要指明第几行。 若dp行数不能省略。 那么只能从1开始。
    for (int i = 0; i < n; i++)
        for (int j = V; j >= w[i]; j--)
            f[j] = max(f[j], f[j - w[i]] + v[i]);
```

1. 恰好装满(也即凑够k元问题)：f[0] = 0, f[1...V]= 负无穷 使用f[0][j]：表示前0个物品，放入容量为j的背包，恰好装满，可以装的最大价值。 问题就是0个物品，对于容量大于0的背包，都不可能装满。所以对于j>0的状态都是未定义的状态。 后序状态不可能由未定义状态转换而来。 所以为了确保 max选择时不会选到这些非法状态， 讲 f[1...v] = 负无穷

2. 最多装载(不要求装满)： f[0...V] = 0 不存在上述的非法状态， f[0][j]：表示前0个物品，放入容量为j的背包，不要求恰好装满， 所以，没有这些非法状态，0个物品，最大价值就是0. 所以 全部赋值0即可。 f[0...V] = 0

3. 总结： 1.滚动数组，倒着算。依赖上一行，不依赖本行 2. 初始化，非法状态，要确保选不到。

完全背包 有N种物品和一个容量为V的背包，每种物品都有无限件可用

```
    dp定义:  f[i][j] = max( f[i - 1][ j], f[i][j - w[i]] + v[i])
    f[i][j]表示前i件物品，可以重复选用的情况下，放入一个容量为j的背包可以获得的最大价值。
    可以发现， 由01背包的f[i - 1][j - w[i]] + v[i] 变为了 f[i][j - w[i]] + v[i]

    for (int i = 1; i <= n; i++)
        for (int j = w[i]; j <= V; j++)
            f[j] = max(f[j], f[j - w[i]] + v[i]);

    **for:j**  :  j变为正序， 此时 f[j]将会依赖于本行左侧的元素。 本行左侧代表，可能已经选用物品i,现在重复选用。
```

LIS: 最长上升子序列

Q300：最长递增子序列

0. 二分法只能找严格递增的子序列。(可以改，二分是，我们去刷新第一个大于num的位置，不能刷新等于的位置)

```
修改为: if(tails[m] <= num) i = m + 1;
```

1. tails[k] 的值代表 长度为k+1 子序列 的 最小 尾部 元素值。
2. 长度为k+1的子序列的最小尾部元素 必然 大于 > 长度为k的子序列的最小尾部元素。

```
        int res = 0;
        // 对于每一个num 更新tails数组
        for(int num : nums) {
            // 在tails数组的 0~res之间二分查找 tails[k] >= num
            // 找第一个大于等于num的位置。
            int i = 0, j = res;
            //  若存在则更新位置i ,若不存在，则i == res 更新的就是res 的位置。

            while(i < j) {
                int m = (i + j) / 2;
                if(tails[m] < num) i = m + 1;
                // m的位置小于num，说明第一个大的一定在m的右侧。
                else j = m;
                // m的位置大于等于num  ，则第一个大的一定在左侧或则就是m.
                // 若m左侧元素都小于num,则最终一直更新i到现在j=m的位置。
                // 若左侧元素等于num，则j一直左移动，直到最左边等于num的位置。
            }
            // 若不存在，则更新的是 tails[res] 下边会使得res+1.
            tails[i] = num;
            // 若res == j //
            说明在 [0~res)中不存在 tails[k] > num 则序列长度+1.
            if(res == j) res++;  // res == i也一样
            // 这里有一个问题，就是该判断使得，该代码只能找 严格递增的，出现相等的，j的位置是在(0~res)之间，则res不会增加。
        }
```

Q334 : 递增的三元子序列

1. Q300的简化。k = 3即可。设一个dp[2]数组即可。当出现num>dp[1]时，dp[1]:说明现在已经有长度为2的子序列了。
2. 而num>dp[1]说明，num不能更新dp[1],应该更新dp[2],说明存在长度为3的子序列了。而dp[2]不需要更新，直接返回true即可。
3. 若一直无法更新dp[2]的位置，说明不存在长度为3的序列。

Q338：比特位计数 对于输入num,给出 0 <= i <= num的 每一个i的比特位数。

1.如： 输入 5 返回 [0,1,1,2,1,2]

2. 原理： 10 : 1010 8： 1000 10的比特位数 恰好是 8的比特位数+1， 即差最后一位。

3. i&(i-1)： 可以消去最后一位1，这样就可以通过10找到8.

```
    for(int i = 1; i <= num; i++){
        dp[i] = dp[i&(i-1)]+1;
    }
```

Q354：俄罗斯套娃信封问题

1. 说白了就是二维的LIS问题，即最长递增子序列。 但是由于二维问题。会互相牵制。
2. 先将数组按照一个维度排序。比如按照w排序。那么直接找 h的最长递增子序列即可。
3. 问题：因为是二维的，如果只按照w排序， 那么w相同的情况下，如 *23 24*，那么使用过2*3后，34是递增的，但是2和2相同
4. 会同时用到 *23，和*24，照成错误。按照我们的想法是，w相同的只能选一个，必然是选h最小的。
5. 由上问题，我们在w排序的基础上，对h进行逆序排序。 即 *24 在 23*前边。 这样按照 二分的LIS算法，3小于4，那么同一个最短子序列就会最终使用 *23将*24覆盖掉。

净胜分问题

Q486：预测赢家： 分数数组： [1, 5, 2] ab两人交替从数组两端选择一个拿走。 最终看a是否能胜利。

1. 假设现有数组[a,...,z]玩家1先手。 那么玩家1可以选a或者z.于是问题被缩小为 选a剩下[b,...,z],或者 选z剩下[a,...,y]
2. 引入 净胜分概念， 即玩家x先手在当前数组中可以获得的最大分数(玩家x获得分数 - 玩家y获得分数)
3. 设dp[i][j]: 表示: 玩家x先手在数组中可以获取到的净胜分。

```
 那么 dp[i][j] = max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1])
 其中 dp[i + 1][j]: 由于自己已经选择了i，那么就相当于对手 在i+1到j是先手的。对手先
手在i+1到j可以获得的最大净胜分。
 因为是对手获得的 净胜分，所以要用减法。

 for (int i = 0; i < len; i++) {
     dp[i][i] = nums[i];
 }
 for (int j = 1; j < len; j++) {
     for (int i = j - 1; i >= 0; i--) {
```

```
        dp[i][j] = Math.max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j -
1]);
    }
}
// 填对角线。
```

LCP19：树叶转色。

0. 假设树叶只有 r红色，y黄色。 给出一个只包含ry的字符串。假设ry可以互相换色。问最少多少次可以将字符串变为[红..黄...红...].每部分最少一个。

1. 使用动态规划。

```
替换成全红排列0: 所需最小替换次数: f[i][0]=f[i-1][0]+isYellow(i)
替换成红黄排列1: 所需最小替换次数: f[i][1]=min{f[i-1][0],f[i-1][1]}+isRed(i)
替换成红黄红排列2: 所需最小替换次数: f[i][2]=min{f[i-1][1],f[i-1]
[2]}+isYellow(i)

第1个树叶，没有红黄，红黄红的状态
第2个树叶，没有红黄红的状态。

// 第一个树叶初始化。
states[0][0]= leaves.charAt(0)=='y'?1:0;
states[0][1]=states[0][2] = Integer.MAX_VALUE;

// 第2个树叶初始化
states[1][2]=Integer.MAX_VALUE;

for(int i=1;i<leaves.length();i++){
    int isYellow= leaves.charAt(i)=='y'?1:0;
    int isRed= leaves.charAt(i)=='r'?1:0;
    //全部替换成红叶的最小次数
    states[i][0]=states[i-1][0]+isYellow;
    //替换成红黄排列最小次数
    states[i][1]=Math.min(states[i-1][1],states[i-1][0])+isRed;
    if(i>=2){
        //替换成红黄红排列的最小次数
        states[i][2]=Math.min(states[i-1][1],states[i-1][2])+isYellow;
    }
}
```

NC127：最长公共子串 LCS

1. ss1[i] == ss2[j]相等时，当j==0时，只能为1， 否则就是 dp[i][j] = dp[i-1][j-1] + 1;

```
int maxi = 0;  int maxlen = 0;
for(int j = 0; j < len2; j++){
```

```
        if(ss1[0] == ss2[j]) {
            dp[0][j] = 1;
            maxlen = 1;
        }
    }
    for(int i = 1; i < len1; i++){
        for(int j = 0; j < len2; j++){
              if(ss1[i] == ss2[j]){
                  if(j == 0){
                       dp[i][j] = 1;
                  }else{
                      dp[i][j] = dp[i-1][j-1] + 1;
                  }
                  if(dp[i][j] > maxlen){
                          maxi = i;
                          maxlen = dp[i][j];
                  }
              }
        }
    }
    //返回子串，根据 maxi,maxlen直接返回字串即可。
    return str1.substring(maxi - maxlen+1, maxi+1);
```

2. dp[i][j] = dp[i-1][j-1] 可以优化空间。 j倒着算即可。

```
    for(int i = 1; i < len1; i++){
        for(int j = len2-1; j >= 0; j--){
              if(ss1[i] == ss2[j]){
                  if(j == 0){  // dp[i][0]单独计算
                       dp[j] = 1;
                  }else{
                      dp[j] = dp[j-1] + 1;
                  }
                  if(dp[j] > maxlen){
                          maxi = i;
                          maxlen = dp[j];
                  }
              }else{
    // ss1[i] != ss2[j] 显式置 0 . 否则 dp[i][j] = dp[i-1][j].不会变为0.会依赖i-2
  行的值。
                  dp[j] = 0;
              }
        }
    }
```

NC92：最长公共子序列： LCS

0. s串和t串匹配。

1. dp[i][j]: 表示：s(0,i),t(0,j)这两个前缀的最长公共子序列。不要求以i,j处的字符结尾。

2. 转移方程：

```
if(ss1[i] == ss2[j])
    dp[i][j] = dp[i-1][j-1] + 1;
else
    dp[i][j] = Math.max(dp[i-1][j],dp[i][j-1]);
```

3. 初始化： 我们假设 s[0]==t[0],那么dp[0][0] = 1 , 则dp[0][1]呢? dp[0][1]一定为1. 因为 s(0,0),t(0,1)的公共子序列就是s[0],长度为1.这说明第一行一列初始化时具有延续性，若前边出现一个1，后边全是1.

4. 如何从dp数组找 公共子序列? :

```
1. dp[i][j] 由 dp[i-1][j-1], dp[i-1][j], dp[i][j-1]三个方向转化而来。
2. 若 dp[i][j] == dp[i-1][j] 就说明, ss1[i]不要, 我们一定能在0-i-1 和0-j找到目标串
3. dp[i][j] == dp[i][j-1] 同理
4. 如果不是上边的情况，则一定是由dp[i-1][j-1]而来。则ss1[i]和ss2[j]就必须匹配

dp[0][0] = ss1[0] == ss2[0] ? 1 : 0;
for (int i = 1; i < len1; i++) {
    dp[i][0] = Math.max(dp[i - 1][0], ss1[i] == ss2[0] ? 1 : 0);
}
for (int i = 1; i < len2; i++) {
    dp[0][i] = Math.max(dp[0][i - 1], ss1[0] == ss2[i] ? 1 : 0);
}
for(int i = 1; i < len1; i++){
    for(int j = 1; j < len2; j++){
        if(ss1[i] == ss2[j])
            dp[i][j] = dp[i-1][j-1] + 1;
        else
            dp[i][j] = Math.max(dp[i-1][j],dp[i][j-1]);
    }
}
if(dp[len1-1][len2-1] == 0) return "-1";

int ii = len1-1;
int jj = len2-1;
int ansl = dp[ii][jj];
char[] ans = new char[ansl];
while(ansl > 0){
    if(ii > 0 && dp[ii][jj] == dp[ii-1][jj]){
        ii--;
    }else if(jj > 0 && dp[ii][jj] == dp[ii][jj-1]){
        jj--;
    }else{
        ans[--ansl] = ss1[ii];
```

```
            ii--;
            jj--;
        }
    }
```

NC91：最长上升子序列：LIS

1. 在 二分法的基础上求出 dp[n] ,在求出dp[n]的同时也求出 maxlen[i]

```
maxlen[i]: 以下标为 i元素结尾的最长子序列。 在求 dp时求出 maxlen
```

2. 从dp可知，最长子序列的末尾元素一定是 dp[ind-1].然后从 arr 数组 合 maxlen数组 求出 ans数组。

```
    arr :  2,1,5,3,6,4,8,9,7   dp = 1,3,4,7,9,0,0,0,0 ind = 5  令j = ind = 5

    arr :  2,1,5,3,6,4,8,9,7
 maxlen :  1 1 2 2 3 3 4 5 4
                         j      9
                       j        8
                     j          4
                 j              3
                ...
    为什么可行呢?  maxlen[i] == j  = 5 :  以9结尾的最长上升子串的长度为 5. 则当
好可以做最后一个元素。因为最长就是5
    然后j--.  再找4.


maxlen[0] = 1;
for(int i = 1; i < n; i++){
    int e = arr[i];
    int left = 0, right = ind;
    while(left < right){
        int mid = left + (right - left) / 2;
        if(dp[mid] <= e){
            left = mid + 1;
        }else{
            right = mid;
        }
    }
    dp[left] = e;
    maxlen[i] = left+1;
    if(right == ind) ind++;
}
int[] ans = new int[ind];
ans[ind-1] = dp[ind-1];
for(int i = n-1, j = ind; j > 0; i--){
    if(maxlen[i] == j) {
```

```
            ans[--j] = arr[i];
        }
    }
```